How Practical are Intrusion-Tolerant Distributed Systems?

Rafael R. Obelheiro, Alysson N. Bessani
Lau C. Lung, Miguel Correia

DI–FCUL TR–06–15

September 2006

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at `http://www.di.fc.ul.pt/tech-reports`. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

How Practical are Intrusion-Tolerant Distributed Systems?*

Rafael R. Obelheiro[1], Alysson Neves Bessani[1], Lau Cheuk Lung[2], Miguel Correia[3]

[1]Departamento de Automação e Sistemas, Universidade Federal de Santa Catarina, Florianópolis, Brazil
[2]Programa de Pós-Graduação em Informática, Pontifícia Universidade Católica do Paraná, Curitiba, Brazil
[3]LASIGE, Faculdade de Ciências da Universidade de Lisboa, Lisboa, Portugal
Email: rro@das.ufsc.br, neves@das.ufsc.br, lau@ppgia.pucpr.br, mpc@di.fc.ul.pt

Abstract

Building secure, inviolable systems using traditional mechanisms is becoming increasingly an unattainable goal. The recognition of this fact has fostered the interest in alternative approaches to security such as *intrusion tolerance*, which applies fault tolerance concepts and techniques to security problems. Albeit this area is quite promising, intrusion-tolerant distributed systems typically rely on the assumption that the system components fail or are compromised independently. This is a strong assumption that has been repeatedly questioned. In this paper we discuss how this assumption can be implemented in practice using *diversity* of system components. We present a taxonomy of axes of diversity and discuss how they provide failure independence. Furthermore, we provide a practical example of an intrusion-tolerant system built using diversity.
Keywords: intrusion tolerance, diversity, security, fault tolerance

1 Introduction

The security of computer systems is constantly challenged by several kinds of attacks, including actions by intruders and various types of malware, such as worms, viruses, and Trojan horses.[1] The defense against these attacks is usually based on preventing intrusions, using methods such as authentication and access control, and appliances like firewalls. However, reality shows that these measures are not entirely effective, and that intrusions are permanently happening. The causes of this situation are complex, but the sheer complexity of current software and the low quality of its development process are conspicuous [26], leaving little hope of effectively preventing intrusions.

An alternative to this preventive approach is *intrusion tolerance* [20], which aims to guarantee that a system works correctly even when some of its parts are compromised [1; 29; 51][2]. The idea is to apply

*This work was partially supported by the EU through project IST-4-027513-STP (CRUTIAL).
[1]See, e.g., the security advisories in the CERT/CC site: http://www.cert.org.
[2]Other expressions have been used in a similar sense, e.g., *resilience*, *survivability* and *distributed trust*.

the *fault tolerance* paradigm — successfully used in critical systems for many years — to the security domain.

Intrusion tolerance usually presumes a distributed system, and underlying this distribution there is an assumption of *failure independence*, that is, that the different system components are not all compromised during a certain window of time. This assumption is quite reasonable when one deals with accidental failures, whether in hardware or in software, since the frequency of such failures is statistically low when the components have a certain quality. In other words, this assumption has a high coverage [40]. However, when one wants to tolerate intrusions, it cannot be assumed that a coordinated attack will not be launched against several components, i.e., that there will be no common mode failures. For instance, most proposals for intrusion-tolerant systems are based on *replication*, either using the state machine approach [44; 6; 5; 12] or quorum systems [33; 56] — see Figure 1. Both approaches replicate services on a number of servers, and the system as a whole is guaranteed to remain correct and operational even if there are intrusions in some of those servers. However, if all replicas are identical and have the same vulnerabilities, then an attack that is effective against one replica is effective against all of them. The solution for guaranteeing failure independence is to use *diversity*: replicas are different — or diverse — thus they do not have the same vulnerabilities[3].

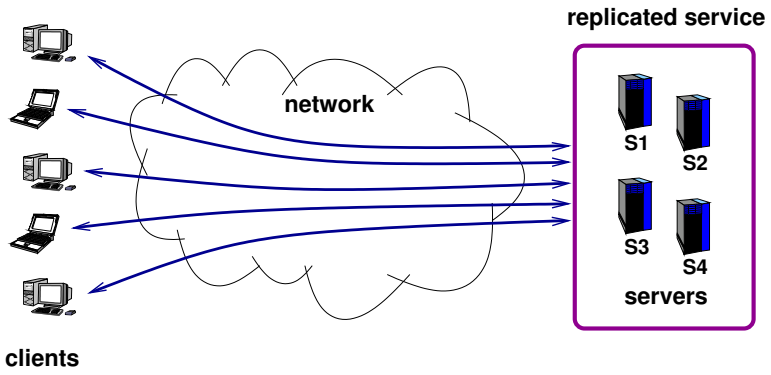

Figure 1: A service replicated to tolerate intrusions in some of the servers.

Prior work dealing with intrusion tolerance usually assumes implementation diversity so that components fail independently [42; 33; 5; 56; 6; 12]. Although this is a common assumption, and a reasonable one from the standpoint of studying intrusion-tolerant protocols and mechanisms, to our knowledge there is little research on the *feasibility of this assumption* for implementing practical systems.

[3]Interestingly, this issue has to do not only with intrusion tolerance but also with the current debate on the importance of diversity to mitigate the problem of worms in the Internet [22]. The problem derives from the current (almost) monoculture of Windows and Microsoft applications in the Internet.

This paper aims to fill this gap, investigating the existing types of diversity and assessing their applicability to the implementation of intrusion-tolerant systems. The contributions of this paper include the introduction of the concepts of *axis* and *degree of diversity*, as well as a taxonomy and an examination of various axes of diversity that can be used for implementing intrusion-tolerant systems. We also provide an example of how these axes can be applied to the design and construction of such a system.

Failure independence is clearly not the only obstacle for building intrusion-tolerant systems. Another important factor is performance, since techniques used in intrusion tolerance are often computationally expensive. However, this issue has been being addressed in other places [6; 35], so we will not expand on this subject in this paper. Other factors are cost and determinism. Both are also out of the scope of this paper, although we briefly discuss the latter when we present the example.

Related Work Design diversity is a classical mechanism for fault tolerance introduced in the 1970s [41]. N-version programming is a technique for creating diverse software components introduced also in those early years [2]. The main idea behind this mechanism is to use N different implementations of the same component, implemented by N different teams of programmers, ideally using different languages and methodologies, to achieve fault tolerance, assuming that designs and implementations developed independently will also fail independently. A good survey of the area can be found in [31]. All these works consider only accidental faults, not attacks/intrusions.

The seminal work on using diversity to improve security is due to Joseph and Avizienis [27], to the best of our knowledge. The paper, however, does not focus so much on diversity but on using diverse components to detect the presence of viruses. Later, Forrest and colleagues applied notions from biologic systems to computer security and argued that diversity is an important natural mechanism to reduce the effects of attacks [19; 25]. Randomized compilation techniques to automatically create diversity in applications were proposed but not developed.

The work most similar to the present paper is the taxonomy of diversity techniques presented by Deswarte et al. [16]. Their taxonomy is defined in terms of the *level* at which diversity is defined: users and operators, human-computer interfaces, application software, execution, hardware and operating system. The taxonomy presented in the current paper is defined in terms of the *component* in which diversity can exist or be created. This taxonomy is more detailed, since we want to clearly identify *where and how* diversity can be obtained. More recently, an important study on diversity in the security domain was presented by Littlewood and Strigini [32]. The paper discusses the reasonability of using diversity

for security and provides an analysis of the probability of failure. The paper takes a quite different approach from ours, since it looks at diversity of components (servers in our architecture), while we investigate in which components of those servers diversity can be created, and how.

Contributions The contributions of the paper are twofold. On the one hand, it proposes a taxonomy that allows designers to identify where and how they should — or must — introduce diversity in intrusion-tolerant distributed systems. On the other hand, it uses an example application to show that diversity can indeed be used on intrusion-tolerant systems.

Paper Organization The rest of this paper is organized as follows: Section 2 presents some aspects of intrusion tolerance. Section 3 introduces the notions of axis and degree of diversity, and presents the taxonomy. Section 4 shows a case study that illustrates a critical web service designed with diversity in mind. Finally, Section 5 presents conclusions and perspectives for further work.

2 Intrusion Tolerance

An intrusion-tolerant system is one that is capable of providing a secure service in a continuous manner in spite of intrusions in a given number of its components [20; 51]. This concept admits a degradation in the functionality offered by the system provided that its security is maintained. Intrusion tolerance is a broad concept that is not restricted to distributed systems, although in practice much of the more interesting work done in the area is precisely on solutions for intrusion-tolerant distributed services [1; 29]. The security properties envisaged are usually availability and integrity, but sometimes also data confidentiality.

The starting point for building an intrusion-tolerant system is eliminating single points of failure, that is, components that may compromise the entire system if their security is broken. This implies the distribution of the system's information and functionality. Coordination across different system components can be accomplished using Byzantine fault-tolerant protocols [30], which are capable of dealing with components subject to arbitrary failures. As already discussed in the introduction, such protocols generally assume that these components fail or are compromised independently. These protocols can be used to handle malicious events, such as *attacks* and *intrusions*, since these events have been shown to be *malicious faults* that aim to activate *vulnerabilities*, which are in turn *design faults* [1; 51].

Cryptographic keys, which are central to several security mechanisms, can also constitute single points of failure, with respect both to confidentiality (when a secret or private key is disclosed to an unauthorized party) and to integrity (when a secret or private key is destroyed without proper authorization). Threshold cryptography [23; 14] alleviates this problem, enabling a secret datum (such as a private or secret key) to be securely shared, increasing its availability without necessarily compromising its confidentiality. In a threshold cryptographic scheme, a secret is divided into a number of parts (called shadows) that are subsequently distributed; access to a subset (with a given minimum size) of these shadows provides the functionality of the original secret. Threshold cryptography has two basic variants. One is secret sharing, where a secret is split into a number of shadows that are distributed across the system; the original secret is later reconstructed from a subset of these shadows. The other variant is secure multi-party computation or function sharing, where the original secret is never reconstructed. Instead, a cryptographically secure function f is computed in a distributed manner: each system component computes f using its own shadow and the results of (a subset of) the various computations are combined to obtain the intended result (e.g., a digital signature).

3 Diversity

The next subsections examine different forms of implementing design diversity in real systems. Our emphasis is on how to obtain failure independence of the components of a distributed system with the goal of making this system more resilient to intrusions. For each individual technique we present its relevance and main benefits together with its eventual drawbacks and how it impacts a system in terms of cost. The discussion is based on two important concepts:

Axis of diversity: a component of a system that may be diversified, i.e., that admits several different instances.

Degree of diversity: the number of choices available for a specific axis of diversity.

For instance, if a system requires the use of a database management system (DBMS), the DBMS is an *axis of diversity* in the system design. If there are three different DBMSs that satisfy the system's requirements, we say that the *degree of diversity* of this axis for the system is three. The taxonomy of axes of diversity is presented in Figure 2.

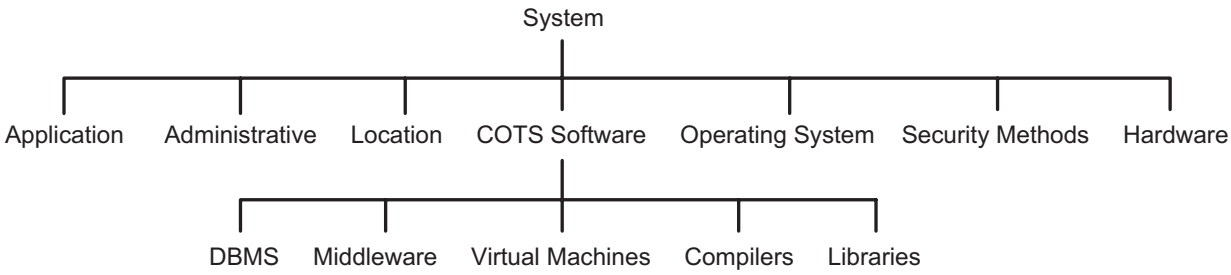

Figure 2: A taxonomy of axes of diversity.

3.1 Application

Diversity of application software is the most usual manifestation of design diversity. The reason is that application software is often the only component over which an organization has total control, possibly having access to its specification and design details. This type of diversity can be obtained using the previously mentioned N-version programming methodology [2]. The techniques suggested by Forrest et al. [19] are also mostly to be used on applications: reordering of application code; reorganization of the memory layout of applications; modifications to process initialization.

A drawback of application implementation diversity is the cost of implementing different variants of a software. However, this cost does not grow in a linear fashion: studies have shown that implementing a variant costs around 70–80% of the cost of the initial variant [16]. This reduction is expected, since requirements elicitation and analysis, a costly phase of the software development process, can be reused for all variants. The black-box tests of the application can be used for all variants as well. There have been several studies on N-version programming, including a (somewhat controversial) study that concluded that there is some correlation among bugs found in different variants, so failure independence is not necessarily guaranteed by this technique [28].

3.2 Administrative

It is a well-known fact that many security compromises are perpetrated through social engineering [54]. Distributing the components of an intrusion-tolerant service across several administrative domains seeks to reduce that problem and to hinder the use of social engineering by an intruder.

This axis of diversity puts different systems under the responsibility of different administrators, which may apply different security management policies. Such policies encompass several issues: software used for protecting the local security domain, configuration and placement of firewalls and intrusion detection systems, as well as how these domains are organized and which security policies apply to their

users. The importance of having different administrators was already pointed out for systems based on Fragmentation-Redundancy-Scattering in [15].

3.3 Location

Location diversity consists of placing several physical components of a system in different sites. That distribution is an important defense against physical threats, either of a malicious nature (theft or destruction of hardware or electrical infrastructure, for instance) or an accidental one (the so-called "acts of God", such as floods, earthquakes, fires, dangerous animals[4] or even a maid that pulls the power plug of a server to connect a vacuum-cleaner...[5]). Natural disasters are, in the vast majority of cases, isolated events, which ensures failure independence. On the other hand, systems that are so important that coordinated attacks against several installation sites must be considered, require stringent physical security policies. This axis of diversity has been acknowledged to be a requirement for the DNS service [9].

It is evident that maintaining a number of adequate physical facilities to accommodate computing systems has a cost. However, often an organization that has multiple administrative units already has such facilities, or it can adapt existing facilities without excessive expenses. Furthermore, location diversity can be combined in a synergetic manner with administrative diversity (Section 3.2). Components located in different places and administered by different people tend to present greater failure independence, and the conjugation of these measures helps to rationalize the costs of adopting these two axes of diversity.

3.4 COTS Software

For a few decades now, systems are rarely built from scratch: nearly every one of them uses some kind of commercial off-the-shelf (COTS) software components[6], whether in the development process (as in the case of compilers and routine libraries) or for implementing large subsystems (such as database management systems or middleware).

The use of COTS components offers a good opportunity for applying diversity, since there are often several components available that implement a certain set of functionalities. Among the numerous types of COTS components commonly used we can consider the following:[7]

[4] One of authors once witnessed a server lost its hard disk after a cockroach laid its eggs on the disk controller.

[5] Also a true story.

[6] In this context, free and open-source software can be considered to be COTS components, even if not strictly commercial.

[7] Operating systems can also be considered COTS components, but they are discussed separately in Section 3.5 due to their importance.

DBMSs Several DBMSs support standardized application programming interfaces (APIs) and connectivity plugins. For instance, the JAVA platform supports the Java Database Connectivity (JDBC) API, which allows the transparent integration of any database that accepts SQL commands (provided that this database has a driver available). Nearly all of the most popular databases have JDBC drivers (some have even more than one, providing diversity also at this level), and they can be used by applications based on that technology as long as the queries and updates to the database follow the SQL-92 standard (supported by all modern relational databases);

Middleware Distributed systems often use middleware platforms to hide the complexity of the interactions among their parts. Some of the most currently used middleware platforms are Web services, the Common Object Request Broker Architecture (CORBA) and Remote Procedure Calls (RPC). These platforms are based on standardized specifications that guarantee interoperability across implementations from different vendors. As middleware mediates all communication among the parts of a system, vulnerabilities in this component effectively compromise the integrity of applications. Therefore, using diverse middleware implementations in the components of a system allow this system to tolerate faults in some of them. A representative example of how this axis of diversity can be easily obtained is the CORBA platform. The Object Management Group (OMG) defines a series of CORBA specifications which are independently implemented by several organizations. Considering only the Java and C++ programming languages, there are at least four high quality, free implementations of this standard (JacORB, OpenORB, TAO, and MICO).

Virtual Machines Languages such as Java, C#, Python, and LISP (among others) execute applications inside a virtual machine (VM). A VM not only manages almost all operating system resources (memory, files, sockets, etc.) used by applications written in those languages, but it also implements features that are fundamental to the integrity of applications (e.g., security monitors). Therefore, VM-related problems can also be considered critical to a system and once again intrusion-tolerant applications can benefit from this axis of diversity. Using Java as an example, we have the Java Virtual Machine (JVM), which (among other things) manages memory, translates bytecodes into native code, and implements security policies using a sandbox. As with CORBA, several quality JVM implementations from different vendors are available [21]. It is worthy noting that several vulnerabilities have been reported in JVMs since Java was introduced.

Compilers Compilers are not part of an intrusion-tolerant application *per se*, but they are fundamental tools for building essentially every application. Due to their internal differences, distinct compilers may produce different object or intermediary code from the same source. Therefore, using compiler diversity provides at least two immediate benefits: *(i.)* it prevents an application from being completely compromised if a compiler generates object code with security vulnerabilities[8], and *(ii.)* for a vulnerability introduced by the programmer, code generated by different compilers might not be affected in the same way by a single exploit[9], as code generation defines, for instance, the order of parameters in the stack. Another compiler-related threat is that of a Trojaned compiler which may insert trapdoors in the code it compiles, an attack publicized by Ken Thompson in his 1984 Turing Award lecture [49]; recent work by Wheeler [53] describes how diversity can help in thwarting this attack.

Libraries Large parts of applications nowadays are not implemented by their developers, who rely on third-party routine libraries such as the standard C library. Therefore, using diverse libraries may provide failure independence with respect to flaws in a particular library. There are many sorts of libraries available, some of which are accessed via standardized interfaces and others through proprietary APIs. The former can easily support diversity simply by using a different implementation of a given interface — as in the case of the Java API for XML Processing (JAXP) and Java Cryptography Extension (JCE) specifications for the JAVA platform — while the latter may be accessed through adapters.

Besides using different COTS components in different parts of an intrusion-tolerant system, one can apply different configurations to these COTS components, for instance, using different options for Just-In-Time (JIT) compilation or using different garbage collection algorithms in virtual machines.

Diversity of COTS components is probably the axis of diversity that has the better cost/benefit relation for a critical system. When compared to other axes, it is inexpensive because COTS components commonly used in the applications, such as DBMSs and compilers, can be obtained with relative easiness (what also makes possible to obtain a high degree of diversity), and it is efficient because too often systems are compromised not through vulnerabilities in the applications themselves, but through vulnerabilities in COTS components used by these applications.

[8] Although code generation with bugs is not so frequent, it is not unheard of, especially with some code optimizations. An example is the `-frename-registers` optimization in the GNU Compiler Collection (GCC), disabled by default in recent versions of the compiler exactly for generating buggy code [47, p. 80].

[9] An exploit is an automated tool that uses specific vulnerabilities with the intention of compromising the security of a system.

Using different variants of a component incurs in an administrative cost, since not all components are administered in the same way. Another issue concerns the interoperability of COTS components. Application developers must ensure that the features they need are available in all variants they expect to use. For instance, if a system requires a DBMS which will be diversified, the developers have to use a set of SQL commands that are available in all variants, avoiding proprietary extensions.

3.5 Operating System

The operating system (OS) plays a fundamental role in the security of a system. It controls all the machine's resources therefore if it has a vulnerability, even if the application has no design or implementation fault, all the system can be vulnerable. The rationale is that applications deal with abstractions created and managed by the OS — files, processes, memory segments — so a vulnerability might allow an attacker to manipulate these abstractions freely. For this reason, a vulnerable OS is the Achilles' heel of a system, irrespectively of the robustness of the software running on top of it. Moreover, current OSs are large and complex and the number of design faults is believed to be proportional to this complexity[10]. Therefore, this axis of diversity is critical to guarantee the independence of machine failures. We are aware of a single intrusion-tolerant system that really uses diversity, the Joint Battlespace Infosphere (JBI) [48]. This system uses four different operating systems: SELinux, Solaris, Windows XP and Windows 2000.

The implementation of OS diversity is facilitated by the existence of standard APIs like IEEE POSIX (Portable Operating System Interface). A standard API make porting an application to a different OS easier, reducing costs and making viable a higher degree of diversity. However, if an API itself has a semantic vulnerability, like allowing a race condition [4], it is probably exploitable in all OSs that implement this API.

OS diversity has a cost in terms of administration personnel. Each different OS requires a specialized administrator that can configure it to be as secure as possible. In fact, not having specialized personnel is probably worse that not using diversity since the overall system may become more vulnerable instead of less vulnerable (the same applies to other COTS components like DBMSs).

Some OSs provide a solution of compromise that guarantees a certain failure independence without requiring different OSs. Available evidence tells that the most common type of attacks in the Internet

[10]Hoglund and McGraw say that Linux and Windows XP have respectively 1.5 and 40 million lines of code, while the bug rates in this kinds of systems vary from 5 to 50 bugs per thousand lines of code [26].

are *buffer overflows* [13; 8]. Most variations of buffer overflow attacks, including stack smashing and heap smashing, require some knowledge of the memory organization to be successful, something that is surprisingly easy in a conventional OS. However, this requirement of knowing the memory layout lead to a recent approach that makes this type of attacks harder: to randomize the organization of the memory [19; 55; 46]. This forces the attacker to tailor the attack to each target, using a brute force attack to discover where is the memory location it has to overwrite. These attacks can be time consuming and can call the attention to the ongoing attack. Memory randomization techniques are available for OpenBSD [37] and for Linux [38; 55].

A similar technique randomizes the instruction set of the processor in order to prevent attacks that try to inject binary code [3]. The idea is that if each machine had its own instruction set then it would be very hard for an attacker to devise code that would run in that machine. Randomized instruction set emulation (RISE) scrambles each byte of the program code using a function parameterized with a per-program random key; before the program is executed, each instruction is descrambled using the inverse function; an attacker would have to know the key to be able to insert code that executed correctly.

3.6 Security Methods

The diversity of security methods [24] is related to the principle of separation of privilege [43][11]. The idea is to use several security methods to enforce each security attribute, in such a way that the attribute is not violated if a subset of these methods is compromised. For instance, to enforce the confidentiality of a message it is possible to encrypt it twice, with two algorithms and two keys, guaranteeing that the content of the message is not disclosed if one of the algorithms or one of the keys is compromised. Another example is the use of two authentication methods, such as passwords and fingerprints.

The objective of this axis of diversity is to maximize the independence of the redundant methods, the same as for any other axis of diversity. For instance, if two methods m_1 and m_2 are used for authentication, a compromise of m_1 should not help compromising m_2. This consideration is especially relevant for cryptographic algorithms since encrypting the same data twice with two different algorithms sometimes does not improve the security, but reduce it [45].

[11] "Where feasible, a protection mechanism that requires two keys to unlock is more robust and flexible than one that allows access to the presenter of only a single key." [43]

3.7 Hardware

Traditionally, hardware redundancy has not been based on diversity probably because the failures of identical hardware components tend to be independent (think, e.g., of Stratus servers[12]). However, recent hardware bugs provide evidence that this axis of diversity can be important for intrusion tolerance. Examples of hardware bugs with security implications are the F00F Pentium bug, which allows a user to block the processor [11], and a bug in the hyper-threading of Intel processors, which allows the unauthorized disclosure of information by a user with low privileges (e.g., stealing private RSA keys used in the machine) [39].

Besides these design faults, hardware diversity is also important for another reason. Exploits found in the Internet are almost always targeted to a specific hardware. Therefore, if an attacker uses an exploit to attack machines with different hardware architectures, the attack will probably be ineffective, or at least will do no more harm than crash the software running in those machines. For the same reason, hardware diversity can be useful against automated attacks using worms.

3.8 Discussion

Several issues related to the axes of diversity we have presented have to be considered. The first is that there can be some dependence among different implementations of a certain component, something that has an impact in the degree of diversity of that axis. Variants that have one (or more) common components have a lower degree of diversity than completely independent components. For instance, several Java Virtual Machines, all based on Sun's code [21], or several mainboards that use the same chips, have a lower degree of diversity than JVMs/mainboards with nothing in common. The degree of diversity of a certain axis is also affected by factors like cost and availability: the higher the number of variants of an axis that can be used with an acceptable acquisition and maintenance cost[13], the higher the degree of diversity of that axis.

Another aspect to be considered is that diversity can be helpful during the system test and validation phases, since it is possible to compare the information obtained for all the variants. This comparison process can help discovering bugs that otherwise would remain dormant, possibly until the system went to production, and the identification of underspecified aspects, since different developers will probably

[12] `http://www.stratus.com`

[13] The meaning of "acceptable" depends on the application. For instance, what is acceptable for a financial application that handles billions of euros every day is different from what is acceptable for an ISP in a small town.

interpret these aspects in different ways.

Recently, both quorum systems and state machine replication started to be used in combination with *proactive recovery* with the objective of increasing the resilience of intrusion-tolerant systems [56; 6]. The idea is to proactively recover the replicas periodically, removing possible intrusions, thus allowing the system to tolerate more intrusions during its lifetime. More precisely, the usual bound on the number of replicas that can be successfully attacked is improved since it no longer applies to the system lifetime but only to a window of vulnerability. Proactive recovery is effective against several types of attacks, but it only works against "fast" and reproducible attacks like buffer overflows if each replica is recovered to a state with different vulnerabilities. This requires mechanisms to automatically generate diversity. There are some currently available, like memory randomization [19; 13; 8; 38; 55; 46] and instruction set randomization [3], but automatic generation of diversity at all axes of diversity is probably the Holy Grail of the area.

Finally, it is important to understand the limitations of diversity. For example, application diversity provides a safeguard against bugs and vulnerabilities in software; it simply cannot be effective if the requirements used for developing this software are incomplete, inconsistent, insecure, or incorrect.

4 Case Study: Critical Web Service

In this section we consider the design of a critical intrusion-tolerant web service to illustrate the application of diversity in implementing a real distributed system.

The objective here is to design a web service that maintains its availability and integrity even if some of its components are compromised (due to faults, attacks and intrusions). A service with this quality of service could be used, for example, for a flight company booking system, allowing its offices deployed all over world to make reservations and bookings. It is important to note that the described architecture could be employed in several other applications.

The architecture of the web service is presented in Figure 3. In this figure the service implementation is distributed in four locations around the world (different company offices). Each one of these locations is composed by an Web Service interface, that could be accessed through Simple Object Access Protocol (SOAP) [52] and a replica of the flight reservations application. These application replicas synchronize their states using *active replication* [44]. In this model of replication, all service replicas execute all requests in the same order, thus ensuring that they are in the same state after executing each request. The

implementation of this technique requires a *total order multicast*[14] protocol. The distributed algorithms literature presents several efficient Byzantine fault-tolerant protocols that could be used in this setting [6; 34; 57].

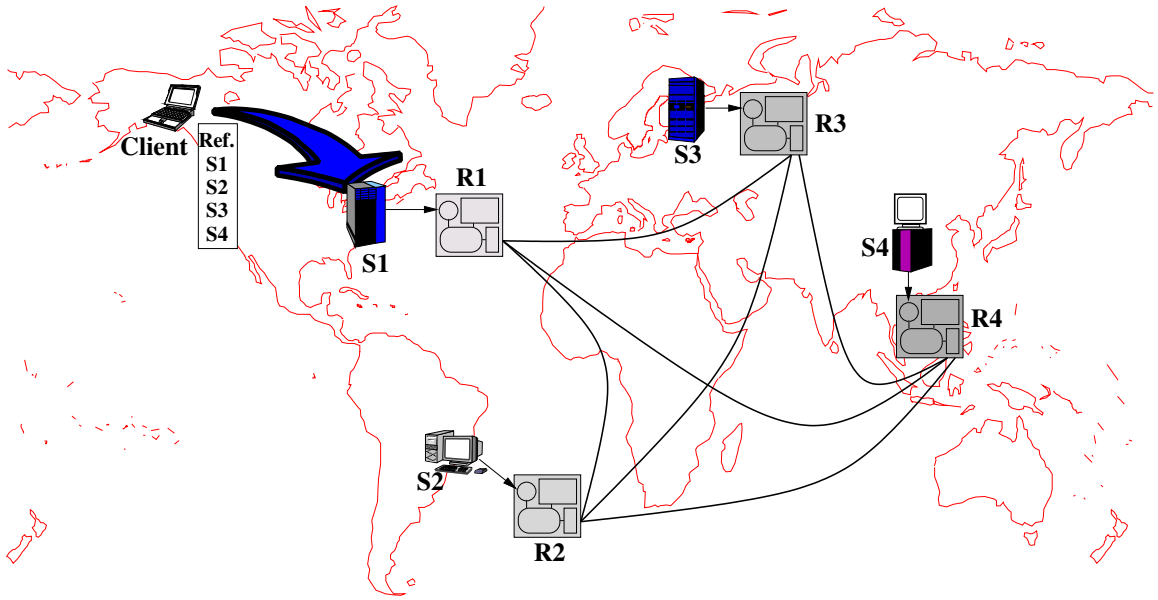

Figure 3: Intrusion-tolerant web services architecture.

A client that wants to access the flight booking system has to retrieve the service description, expressed in the Web Services Description Language (WSDL) [10], registered in some Universal Description, Discovery and Integration (UDDI) service [36] available on the Internet. In this description of the service there is a list with the four web servers that can be used to have access to the booking system. Clients can access the system using any of these servers.

The protocols used to implement active replication (total order multicast) in the Internet require that less than a third of the system replicas be faulty at any instant of time. Therefore, the system depicted in Figure 3 tolerates at most one compromised replica. In order to make faults and intrusions independent in this system we have to employ diversity to implement it.

For this example, we consider six axes of diversity for the four system replicas (R1–R4): implementation, execution environment (COTS), database (COTS), operating system, hardware, and location. Table 1 presents some possible choices for implementing this system, with every axis having degree of diversity equal to four.

Some comments can be made about the Table 1. Firstly, to provide application diversity, four variants of the service software are built: two using JAVA and two using C#. These variants have to be executed

[14] Also called *atomic multicast*.

Axis of Diversity	R1	R2	R3	R4
Application	Java 1	Java 2	C# 1	C# 2
Execution Environment	Sun Java[a]	Free Java[b]	.NET	Mono
Database	MySQL	PostgreSQL	IBM DB2	Firebird
Operating System	Solaris	FreeBSD	Windows	Linux
Hardware	Ultra SPARC	Pentium	Athlon	Mac
Location	USA	Brazil	Russia	China

[a] J2SRE + JWSDK (Java Web Services Development Kit).
[b] Kaffe + GNU Classpath + Apache Axis.

Table 1: Example of usage of diversity for a critical Web service.

in different execution environments each comprising a virtual machine and a web services middleware. Another interesting aspect of the presented example is that the location diversity in practice implies administrative diversity, since it is unlikely that sites so far apart will be managed by the same personnel.

The application to be developed following the design proposed in this section will be intrusion-tolerant and will maintain failure independence using mostly components already available. This is possible because vulnerabilities in any of the components used in individual replicas of the system can be exploited only in a single replica. A single social engineering-based attack cannot affect more than one replica due to location/administration diversity. Location diversity also provides some resistance against physical attacks to the computing and communications infrastructure in addition to limited resistance against denial of service attacks.

4.1 The importance of number four

One must notice that the system proposed in this section is composed of four replicas and that for each diversity axis, we consider a degree of diversity of four (four variants). This choice is not arbitrary: Byzantine fault-tolerant agreement protocols (such as total order multicast) generally tolerate f faults if the number of replicas is $n \geq 3f + 1$ [30; 50] (an exception is [12] that requires only $2f + 1$ replicas). Therefore, the minimal number of replicas that a system must have is four (to tolerate one fault). In order to maintain the maximum degree of failure independence, the architect of an intrusion-tolerant system must define a degree of diversity of at least four for each axis of diversity, and ideally this degree must be equal to the number of replicas n. For example, if in the example we had used two replicas running Windows, some vulnerability in this system could compromise these two replicas. This situation could destroy the whole service since its coordination protocol tolerates only a single failure.

4.2 Determinism

Active replication requires that service replicas are deterministic, i.e., that the same command executed in the same initial state generates the same final state irrespectively of the replica in which it is executed. A recent experience in replicating a web service has shown how determinism is far from trivial to obtain [17]. Only static HTML pages were considered (no PHP or ASPs) and the communication protocol was only HTTP (no HTTP-S). All replicas were identical (Linux with Apache servers). Even with such restrictions and such a nice environment, HTTP headers have several fields that impair determinism [18]. Three fields were problematic: the timestamp with the date and time in which the page was returned (it is not exactly the same in all servers); the field *Etag* that univocally identifies a reply to an HTTP request; and the server identifier.

A partial solution to enforce determinism is provided in BASE [7]. An abstract specification of the service, its state and of operations that manipulate the state is provided. Then a conformance wrapper is defined that interfaces the requests with each (different) replica. If state transfers are necessary, then a function that translates from the state of a replica to the abstract state and vice versa has also to be defined. This scheme, however, cannot handle all sources of indeterminism, like timestamps, compressed or encrypted data.

5 Conclusions

The study presented in this paper allow us to conclude that it is *possible* to implement intrusion-tolerant systems in *practice* if diversity is applied correctly. Diversity comes at a cost, derived among other factors from the extra complexity introduced, but many critical systems have security requirements that may justify this cost.

A contribution of this work is the introduction of the concepts of axis and degree of diversity, and the study of a taxonomy of axes that can be used to implement intrusion-tolerant services. A second contribution is the proposal of a simple and effective architecture for implementing practical intrusion-tolerant web services in the Internet using already available COTS.

The work presented in this paper opens interesting questions regarding the use of diversity in implementing intrusion-tolerant systems. In particular, it seems fundamental to examine the systematic use of diversity [7], especially when supported by tools and methods, in a way that it will be easier to implement failure independence. Another possible future direction for the work presented here is the

implementation of a system like the one presented in Section 4 trying to evaluate the practical difficulties that emerge when implementing a real intrusion-tolerant system (some of these difficulties, like COTS compatibility, we already discussed in this paper). In this context, an open question is what metrics and methods will be more suited to evaluate how dependable this system would be, and what is the cost of such dependability level. The study presented in [32] is a good starting point but it is unclear how the decomposition of servers in several components (hardware, OS, application code, COTS components) will impact the formulas presented.

References

[1] A. Adelsbach, D. Alessandri, C. Cachin, S. Creese, Y. Deswarte, K. Kursawe, J. C. Laprie, D. Powell, B. Randell, J. Riordan, P. Ryan, W. Simmonds, R. Stroud, P. Veríssimo, M. Waidner, and A. Wespi. *Conceptual Model and Architecture of MAFTIA. Project MAFTIA deliverable D21*. Jan. 2002.

[2] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proceedings of the IEEE COMPSAC*, pages 149–155, Chicago, IL, USA, Nov. 1977.

[3] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289, 2003.

[4] M. Bishop and M. Dilger. Checking for race conditions in file access. *Computing Systems*, 9(2):131–152, 1996.

[5] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the Internet. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2002)*, Washington, DC, USA, 2002.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21(3):236–269, Aug. 2003.

[8] S. Chen, J. Xu, Z. Kalbarczyk, and K. Iyer. Security vulnerabilities: From analysis to detection and masking techniques. *Proceedings of the IEEE*, 94(2):407–418, Feb. 2006.

[9] S. Cheung. Denial of service against the domain name service. *IEEE Security & Privacy*, 4(1):40–45, Jan./Feb. 2006.

[10] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language 1.1*. W3C Working Group, Mar. 2001.

[11] R. R. Collins. The Pentium F00F bug. *Dr. Dobb's Journal of Software Tools*, 23(5):62, 64–66, May 1998.

[12] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, Florianópolis, Brazil, Oct. 2004.

[13] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*, Hilton Head Island, SC, USA, Jan. 2000.

[14] Y. G. Desmedt. Some recent research aspects of threshold cryptography. In E. Okamoto, G. Davida, and M. Mambo, editors, *Proceedings of the First International Workshop on Information Security (ISW'97)*, LNCS 1396, pages 158–173, Ishikawa, Japan, Sept. 1997. Springer-Verlag.

[15] Y. Deswarte, L. Blain, and J. C. Fabre. Intrusion tolerance in distributed computing systems. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pages 110–121, Oakland, CA, USA, May 1991.

[16] Y. Deswarte, K. Kanoun, and J.-C. Laprie. Diversity against accidental and deliberate faults. In P. Ammann, B. H. Barnes, S. Jajodia, and E. H. Sibley, editors, *Computer Security, Dependability, and Assurance: From Needs to Solutions*, pages 171–181, Williamsburg, VA, USA, Nov. 1998. IEEE Computer Press.

[17] R. Ferraz, B. Goncalves, J. Sequeira, M. Correia, N. F. Neves, and P. Veríssimo. An intrusion-tolerant web server based on the DISTRACT architecture. In *Proceedings of the Workshop on Dependable Distributed Data Management*, Florianópolis, Brazil, Oct. 2004.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. IETF Request for Comments: RFC 2068, Jan. 1997.

[19] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, pages 67–72, May 1997.

[20] J. S. Fraga and D. Powell. A fault- and intrusion-tolerant file system. In *Proceedings of the 3rd International Congress on Computer Security (IFIP/SEC'85)*, pages 203–218, Dublin, Ireland, Aug. 1985.

[21] D. K. Friedman and D. A. Wheeler. Java implementations. On-line at `http://www.dwheeler.com/java-imp.html`, Aug. 2002. (Accessed 05 Apr. 2006).

[22] D. E. Geer, D. Aucsmith, and J. A. Whittaker. Monoculture. *IEEE Security & Privacy*, 1(6):14–19, Nov./Dec. 2003.

[23] P. S. Gemmell. An introduction to threshold cryptography. *Cryptobytes—The Technical Newsletter of RSA Laboratories*, 2(3):7–12, Winter 1997. `ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto2n3.pdf`.

[24] M. A. Hiltunen, R. D. Schlichting, and C. A. Ugarte. Building survivable services using redundancy and adaptation. *IEEE Transactions on Computers*, 52(2):181–194, Feb. 2003.

[25] S. A. Hofmeyr and S. Forrest. Architecture for an artificial immune system. *Evolutionary Computation*, 8(4):443–473, Dec. 2000.

[26] G. Hoglund and G. McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, 2004.

[27] M. K. Joseph and A. Avizienis. A fault-tolerant approach to computer viruses. In *Proceedings of the 1988 IEEE Symposium on Research in Security and Privacy*, pages 52–58, Apr. 1988.

[28] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, Jan. 1986.

[29] J. H. Lala, editor. *Foundations of Intrusion Tolerant Systems*. IEEE Computer Society Press, 2003.

[30] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[31] B. Littlewood, P. Popov, and L. Strigini. Modelling software design diversity – a review. *ACM Computing Surveys*, 33(2):177–208, June 2001.

[32] B. Littlewood and L. Strigini. Redundancy and diversity in security. In P. Samarati, P. Rian, D. Gollmann, and R. Molva, editors, *Computer Security – ESORICS 2004, 9th European Symposium on Research Computer Security*, LNCS 3193, pages 423–438. Springer, 2004.

[33] D. Malkhi and M. K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11:203–213, 1998.

[34] J.-P. Martin and L. Alvisi. Fast Byzantine consensus. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2005)*, Yokohama, Japan, June 2005.

[35] H. Moniz, M. Correia, N. F. Neves, and P. Veríssimo. Randomized intrusion-tolerant asynchronous services. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'2006)*, Philadelphia, PA, USA, June 2006. Accepted for publication, to appear.

[36] OASIS. *Universal Description, Discovery and Integration v3.0.2 (UDDI)*. Organization for the Advancement of Structured Information Standards (OASIS), Oct. 2004.

[37] The OpenBSD project. `http://www.openbsd.org/`. (Accessed 05 Apr. 2006).

[38] Homepage of the PaX team. `http://pax.grsecurity.net/`. (Accessed 05 Apr. 2006).

[39] C. Percival. Cache missing for fun and profit, May 2005. On-line at `http://www.daemonology.net/papers/htt.pdf`. (Accessed 05 Apr. 2006).

[40] D. Powell. Fault assumptions and assumption coverage. In *Proceedings of the 22nd IEEE International Symposium on Fault-Tolerant Computing*, July 1992.

[41] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, SE-1:220–232, June 1975.

[42] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 99–110. Springer-Verlag, 1995.

[43] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[44] F. B. Schneider. Implementing fault-tolerant services using the state-machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990. Reprinted in S. J. Mullender (ed.), *Distributed Systems*, Second Edition. New York, NY, USA: ACM Press, 1993.

[45] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. John Wiley & Sons, New York, NY, USA, 2nd edition, 1996.

[46] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, pages 298–307, Washington, DC, USA, Oct. 2004.

[47] R. Stallman. *Using the GNU Compiler Collection (version 4.0.3)*. Free Software Foundation, Boston, MA, 2005. On-line at `http://gcc.gnu.org/onlinedocs/`. (Accessed 05 Apr. 2006).

[48] F. Stevens, T. Courtney, S. Singh, A. Agbaria, J. F. Meyer, W. H. Sanders, and P. Pal. Model-based validation of an intrusion-tolerant information system. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, pages 184–194, Florianópolis, Brazil, Sept. 2004.

[49] K. Thompson. Reflections on trusting trust. *Commun. ACM*, 27(8):761–763, Aug. 1984.

[50] S. Toueg. Randomized byzantine agreements. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 163–178, 1984.

[51] P. Veríssimo, N. F. Neves, and M. Correia. Intrusion-tolerant architectures: Concepts and design. In R. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*, LNCS 2677, pages 3–36. Springer-Verlag, 2003.

[52] W3C. *SOAP 1.2 – W3C Recommendation*. W3C, June 2003. `http://www.w3.org/TR/soap12/`. (Accessed 05 Apr. 2006).

[53] D. A. Wheeler. Countering trusting trust through diverse double-compiling. In *Proceedings of the 21st Annual Computer Security Applications Conference (ACSAC)*, pages 28–40, Tucson, AZ, USA, Dec. 2005.

[54] I. S. Winkler and B. Dealy. Information security technology?...don't rely on it—a case study in social engineering. In *Proceedings of the 5th USENIX UNIX Security Symposium*, Salt Lake City, UT, USA, June 1995.

[55] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *Proceedings of the 22nd IEEE Symposium on Reliable Distributed Systems*, pages 260–269, Oct. 2003.

[56] L. Zhou, F. B. Schneider, and R. van Renesse. COCA: A secure distributed online certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.

[57] P. Zielinski. Paxos at war. Technical Report UCAM-CL-TR-593, University of Cambridge Computer Laboratory, Cambridge, UK, June 2004.